Informatyczny kącik olimpijski (133): *Cyfrowy ciąg*

W tym odcinku omówimy rozwiązanie zadania „Cyfrowy ciąg”, które pojawiło się w eliminacjach do zawodów *Romanian Master of Informatics*.

Cyfrowy ciąg: *Tadek napisał ciąg $S = (s_1, s_2, \ldots, s_n)$, złożony z n cyfr od 1 do 9. Chciałby podzielić ten ciąg na k spójnych fragmentów. Każdy fragment czytany od lewej do prawej tworzy liczbę. Tadek chciałby dokonać takiego podziału, aby największa z k otrzymanych liczb była jak najmniejsza.*

Niech $S[a : b]$ oznacza fragment $s_a, s_{a+1}, \ldots, s_b$.

Zauważmy, że S zawiera tylko cyfry od 1 do 9 – nie ma cyfry 0. Zatem dowolny jego fragment opisuje liczbę bez zer wiodących. Od tego momentu zakładamy, że rozważamy tylko liczby bez zer wiodących. Wiadomo, że dłuższa liczba jest większa niż krótsza. W związku z tym chcemy tak podzielić ciąg, aby największa liczba była możliwie najkrótsza. Największa liczba zawiera przynajmniej $\lceil \frac{n}{k} \rceil$ cyfr. Zauważmy, że w jakimś optymalnym podziale będą wyłącznie liczby $\lceil \frac{n}{k} \rceil$-cyfrowe lub $(\lceil \frac{n}{k} \rceil - 1)$-cyfrowe. Dokładniej, w jakimś optymalnym podziale będzie:

- x liczb $\lceil \frac{n}{k} \rceil$-cyfrowych, gdzie $x = k$, jeśli $k|n$ i $x = n \bmod k$ w przeciwnym przypadku,
- y liczb $(\lceil \frac{n}{k} \rceil - 1)$-cyfrowych, gdzie $y = k - x$.

Niech $d_x = \lceil \frac{n}{k} \rceil$ oraz $d_y = \lceil \frac{n}{k} \rceil - 1$. Szukamy takiego podziału S na x liczb d_x-cyfrowych i y liczb d_y-cyfrowych, żeby największa liczba była jak najmniejsza.

Rozwiązanie $O(nk)$
Ten problem możemy rozwiązać za pomocą metody programowania dynamicznego. Otóż niech $DP[i][j]$ oznacza wartość największej liczby w optymalnym podziale $id_x + jd_y$ pierwszych cyfr ciągu na i liczb d_x-cyfrowych oraz j liczb d_y-cyfrowych. Wyznaczymy wartości $DP[i][j]$ dla wszystkich takich i, j, że $0 \leqslant i \leqslant x$ oraz $0 \leqslant j \leqslant y$. Wówczas wynikiem będzie $DP[x][y]$. Niech $DP[0][0] = 0$, zaś $DP[i][j] = \min(p_1, p_2)$ dla $i + j > 0$, gdzie:

- p_1 to wynik podziału przy założeniu, że ostatnia liczba ma d_x cyfr. Wtedy $p_1 = \max(DP[i-1][j], S[1 + (i-1)d_x + jd_y : id_x + jd_y])$, jeśli $i > 0$ i $p_1 = \infty$ w przeciwnym przypadku.
- p_2 to wynik podziału przy założeniu, że ostatnia liczba ma d_y cyfr. Wtedy $p_2 = \max(DP[i][j-1], S[1 + id_x + (j-1)d_y : id_x + jd_y])$, jeśli $j > 0$ i $p_2 = \infty$ w przeciwnym przypadku.

Mamy $O(k)$ liczb „dłuższych” oraz $O(k)$ liczb „krótszych”, czyli wszystkich stanów do obliczenia jest $O(k^2)$. Obliczenie jednego stanu zajmuje $O(\frac{n}{k})$ (porównanie dwóch liczb długości $O(\frac{n}{k})$). Zatem całe rozwiązanie działa w czasie $O(nk)$.

Rozwiązanie $O(\frac{n^2}{k} \cdot \log(n))$
W tym rozwiązaniu wykorzystamy algorytm wyszukiwania binarnego. Wiemy, że wynikiem jest wartość jakiegoś podsłowa długości d_x. W pierwszej fazie rozwiązania weźmy wszystkie d_x-cyfrowe podsłowa i uporządkujmy je niemalejąco. Takich liczb jest $O(n)$. Ich posortowanie wymaga $O(n \cdot \log(n))$ porównań, zaś jedno porównanie zajmuje czas $O(\frac{n}{k})$. Zatem pierwsza faza działa w czasie $O(\frac{n^2}{k} \cdot \log(n))$.

Kiedy mamy już uporządkowany ciąg i wiemy, że jedna z tych liczb jest wynikiem, to możemy wykonać w tym ciągu wyszukiwanie binarne – drugą fazę rozwiązania. Załóżmy, że w kroku algorytmu sprawdzamy, czy wynik jest nie większy niż w. Zatem chcemy dowiedzieć się, czy istnieje podział ciągu na co najwyżej k liczb nie większych niż w. W tym celu będziemy dokonywali podziału od lewej do prawej. W każdym kroku zachłannie wybieramy największą możliwą liczbę. Jeśli liczba tworzona przez d_x kolejnych cyfr jest nie większa niż w, to dodajemy ją do podziału. W przeciwnym przypadku dodajemy liczbę o jeden krótszą. Jeśli po k krokach wykorzystamy wszystkie cyfry, to znaleźliśmy podział o wyniku co najwyżej w.

Algorytm wyszukiwania binarnego wykona $O(\log(n))$ kroków. W każdym kroku przechodzimy po całym ciągu, co zajmuje $O(n)$ operacji. Zatem druga faza działa w czasie $O(n \cdot \log(n))$, a w całym rozwiązaniu wykonuje się $O(\frac{n^2}{k} \cdot \log(n))$ operacji.

Rozwiązanie $O(n \cdot \log(n))$
Spróbujmy przyspieszyć pierwszą fazę poprzedniego rozwiązania. W tym celu wykorzystamy słownik podsłów bazowych, którego konstrukcja wymaga $O(n \cdot \log(n))$ operacji. Za pomocą tej struktury danych możemy w czasie $O(1)$ porównać leksykograficznie dwa podsłowa, co jest równoważne porównaniu wartości liczbowych przez nie reprezentowanych. A więc pierwszą fazę rozwiązania wykonujemy w czasie $O(n \cdot \log(n))$.

Wersja trudniejsza – cyfry na okręgu
Dla ambitnych Czytelników proponujemy trudniejszą wersję zadania – cyfry są ułożone na okręgu, który możemy rozciąć w dowolnym miejscu. Przedstawimy krótki zarys rozwiązania. Zduplikujmy ciąg S, sklejając dwie jego kopie. W ten sposób każde podsłowo długości n opisuje jakąś rotację cykliczną S. Wynik będziemy wyszukiwali binarnie (jak w poprzednim rozwiązaniu). W każdym kroku, dla każdej pozycji wyznaczamy najdalszą pozycję na prawo, gdzie może zaczynać się kolejna liczba. Jeśli pozycje zinterpretujemy jako wierzchołki, a przejścia do kolejnych liczb jako krawędzie, to otrzymamy graf acykliczny. Pytamy, czy w takim grafie istnieje k-krawędziowa ścieżka, która pokrywa n liter. Można to sprawdzić metodą skoków potęgami dwójki.

Bartosz ŁUKASIEWICZ